NDDL Reference

Modeling with NDDL

This page presents the NDDL modeling language for describing problem domains and constructing partial plans. NDDL is an acronym for New Domain Description Language which reflects its origins in DDL[1] which established many of the semantics and constructs found in NDDL. NDDL is an object-oriented language designed to support expression of problem domain elements and axioms in a manner consistent with the paradigm outlined on the Plan Representation page. The reader is strongly advised to review that material prior to working through this chapter.

NDDL is introduced here in something of a bottom up fashion, starting with variables and constraints, moving onto classes of increasing sophistication, and then exploring the expressive language for writing rules. The NDDL language for allocating elements of a partial plan in a plan database is used throughout in support of examples, and detailed more comprehensively at the end of the chapter. Note that many of the code snippets used as examples here originate in the Mars Rover example/tutorial.

NOTE: This page discusses the NDDL language. To understand the NDDL parser and how to use it, see NDDL Parser.

Variables and Constraints

Variables and constraints are the basic building blocks of E2. They can be introduced in a number of ways, and in a range of scopes. The following basic forms for variable declaration are supported:

- *type label*: Declares a variable of type *type* with the name *label* and allocates a default domain as the base domain based on the type. The available primitive types are: *int*, *float*, *bool* and *string*. In addition, a type can be a user-defined enumeration or class. The default base domain for *int* is *[-inf +inf]*, and for *float* it is *[-inff +inff]* i.e. from negative to positive infinity. Variables of type *bool* have a base domain of *{false, true}*. Variables of type *string* are a special case requiring a singleton value on declaration (see our warning).
- *type label* = *value*: Declares a variable of type *type* with the name *label* and sets the base domain to the singleton value given by *value*.
- *type label* = *domain*: Declares a variable of type *type* with the name *label* and sets the base domain to *domain*. For *int* and *float* the domain is described as an interval.

In each of the above cases, the term *domain* reflects the base set of values in the domain of the variable. The variable cannot contain any values not present in this initial domain unless the type of the variable is *open*.[2] This domain is called the *base domain*.

Figure 1 provides examples of different forms of variable declaration and restriction. Notice in particular the use of *enum* and *typedef* keywords which introduce user-defined types. Notice also that *int* and *float* types are restricted to intervals and that the decimal point is only used (and must be used) for *float* values.

```
//Primitives with defaeult base domains
int v0;               // Declares an integer variable v0 with a domain [-inf +inf]
float v1;             // Declares a float variable v1 with a domain [-inff +inff]
bool v2;              // Declares a bool variable v2 with a domain {false, true}

// String requires explicit instantiation
string v3 = "NDDL is Not DDL";

// Declare a symbolic enumeration as a custom type
enum SPEED {SLOW, MEDIUM, FAST};
SPEED v4;             // Declares a variable with an enumeration of values {SLOW, MEDIUM, FAST}

// Use custom type definitions
typedef int [1 10] INT_TEN;
INT_TEN v5;
```

Figure 1: Examples of different forms of variable declaration and restriction

Figure 2 provides a NDDL specification of a simple *Constraint Satisfaction Problem* or CSP. A CSP is given by a set of variables and a set of constraints. The problem is solved when all variables are assigned a value and all constraints are satisfied. In this case, the variables are the integer variables a, b, c, and d. Each is initialized with an initial domain of values. Notice that the constraints a == b; b <= c; b + c == d. These are expressed using the constraints *eq*, *leq* and *addEq* respectively. The solution to the problem is: a {5}, b{5}, c{15}, d {20}.

```
#include "Constraints.nddl"

// Declare variables of type int.
int a; // Declare a variable of type int. It's base domain = [-inf +inf]
int b = [-inf +inf]; // Equivalent declaration but making the base domain explicit.
int c = 15; // In place initialization to a singleton value. Base domain = 40.
int d = [1 20]; // In place initialization to a finite interval. Base domain = [1 20].

// Post constraints on variables
eq(a, b); // a == b
leq(b, c); // b <= c
addEq(b, c, d); // b + c == d
```

```
eq(b, [4 8]); // b == [4 8]
eq(a, 5); // a == 5

// Now we show use of a library constraint to verify expected solutions to the CSP.
testEQ(true, a, 5); // Asserts that a == 5
testEQ(true, b, 5); // Asserts that b == 5
testEQ(true, c, 15);// Asserts that c == 15
testEQ(true, d, 20);// Asserts that d == 20
```

Figure 2: A simple *Constraint Satisfaction Problem* in NDDL illustrating global variable declaration and constraint allocation.

NDDL does not allow arithmetic operators to specify the constraints explicitly. Instead, constraints are drawn from a system or user-defined library of available constraints which are registered under a constraint name. Such names cannot be reserved words in NDDL, but they may otherwise be any alpha-numeric string. The semantics of the relationship imposed by a registered constraint are not defined within NDDL. Rather, these semantics must be defined by the constraint provider. The general form for introducing a constraint is:

- *constraintName(arg0, arg1[, arg2[, ..argN]])*; Allocates an instance of a constraint registered under the name *constraintName*. A constraint has 2 or more arguments. An argument can be one of: *label*, *value*, or *domain*.

For a complete list of available constraints in NDDL, see the Constraint Library Reference.

Classes

A class is a user-defined abstract data type that is used when either:

1. The instances of the type are problem specific. For example, the set of *speeds* might be encoded in the model but the set of *locations* might be problem specific. However, the model will reference *locations* without knowledge or necessity of the instance details.
2. The instances of the type have structure. For example, a *path* might include a *source*, *destination* and *cost*.
3. The instances of the type have state that temporally scoped (i.e. described by predicates).

Basic Definition and Instantiation

The simplest possible class can be introduced in a model thus:

```
class Foo {} // Declare a class Foo. It has no members and no custom constructor.
```

Instances of classes are allocated when setting up a problem. An object (i.e. an instance of a class) is allocated with the imperative *new* operator:

```
new Foo(); // Allocate an object of class Foo. Constructor for Foo has no arguments.
```

There is no NDDL command to delete an object. All objects are uniquely named. If, as above, no name is provided, a unique name will be generated. A variable of a class is declared and instantiated with a base domain of all objects of that class that have been allocated.

```
Foo f; // Allocate a variable whose domain is the set of all objects of class Foo.
```

As with other variable declaration statements, the domain of the variable can be restricted:

```
Foo f1 = new Foo(); // Allocate an object and an object variable. Restrict variable to singleton of new
Foo f2 = f1; // Allocate a variable and restrict its domain to the object held in f1
```

As with any other variables, object variables can participate in constraints:

```
Foo f1;
Foo f3;
eq(f3, f1);
```

It is possible that an object variable is declared before objects of the given type have been allocated. As long as the class is open, this is not a problem, since empty domains are permitted for variables of open types. However, if the class has been closed for a problem, indicating no further instances can be added, and no instances are present, then the variable will be empty on allocation and the system will be considered *inconsistent*. To illustrate this, consider the following NDDL model and problem description

```
// The model has 2 class declarations
class Foo{}
class Bar{}

// The problem instance is allocated procedurally by the following statements
Foo f1; // variable allocated before any objects
Foo f2 = new Foo(); // After this f1 will contain the new object stored in f2.
close(); // Close all types
Bar b; // Base domain will be empty and type is closed. This causes inconsistent initial state.
```

The most common situation is that all objects are allocated and then all the classes are closed using the *close* command without qualification. This is preferred since stronger inference is possible on closed domains. However, for applications where the set of instances may be increased after the initial state is created, it may be important to leave a class open. For example, suppose an application requires target tracking and the set of targets to track is dynamic, evolving throughout execution. As execution progresses, new targets are added and these can be incorporated into the planning system smoothly as long as the type for targets was left open. Later sections will explore the semantics and consequences of dynamic objects (instances of open classes) in more detail.

Composition and Construction

Classes typically have more structure than *Foo* and *Bar* from the previous section. Consider an application involving path navigation. Navigation must deal with path selection from place to place. Different path and location networks may arise in different problem instances, but the basic structure of a path network can be re-used across problem instances. Figure 3 illustrates such a path network structure.

```
#include "Plasma.nddl"

// Declare Location as a class to allow allocation of instances
// at run-time rather than compile time.
class Location {
   string name;
   Location(string _name){
      name = _name;
   }
}

// Declare Path as a triple of locations and cost
```

```
class Path {
   Location from; // Declare member variable 'from' of type 'Location'
   Location to; // Declare member variable 'to' of type 'Location'
   float cost = 0.0; // Declare member variable 'cost' of type 'float' with a default value.

   // Specify a constructor to initialize members
   Path(Location _from, Location _to){
      from = _from;
      to = _to;
   }

   // Another constructor with only one Location - trivial path
   Path(Location loc){
      from = loc;
      to = loc;
   }

   // Another constructor which over-rides default cost
   Path(Location _from, Location _to, float _cost){
      from = _from;
      to = _to;
      cost = _cost;
   }
}
```

Figure 3: a model of a Path Network

Locations are modeled with a class *Location* which includes a string name as a member variable. Note the declaration of a member variable. The body of the class declaration specifies the default base domains for each variable. All manner of declaration introduced for variables can be applied within the scope of a class declaration. For example:

```
class Bar{}
class Foo {
    int arg1; // [-inf inf]
    float arg2 = [0.1 10.0]; // [0.1 10.0]
    bool arg3; // {false, true}
    string arg4 = ?STRING DEFAULT?;     // WARNING:  Cannot be changed to a different value later!
    Bar arg5; // All instances of Bar by default
    Bar arg6 = new Bar(); // A new instance always
}
```

Recall that *string* variables must always have explicit base domains (see our warning). In the case of *Location.name* in Figure 3, since no specific base domain was provided on declaration, NDDL requires a constructor to pass in a specific *name* on allocation. A constructor is invoked using the new operator first introduced in the previous section. For example:

```
Location l1 = new Location(?Hill?);
```

Constructor execution is procedural, meaning it proceeds line by line in the order of declaration. Constructor statements are limited to variable assignment. Notice in Figure 3 that more than one constructor is provided for class *Path*. This supports different modes of initialization. There is no limit to the number of constructors that can be defined for a class. Assignments in a constructor take precedence over default assignments made in member variable declarations. It is common practice to define a model in a separate file from an initial state description. NDDL allows file inclusion as follows:

```
#include ?PathNetwork.nddl?
Location l1 = new Location(?Hill?);
Location l2 = new Location(?Home?);
Path p1 = new Path(l1, l2, 10);
```

The above NDDL fragment allocates 2 locations and a single path between them with a cost of 10. Obviously, more elaborate path networks can be created using this pattern.

Predicate Declaration

Thus far, the discussion of classes has been limited to problem domain elements without any temporal context. Predicates provide the means to describe temporally qualified state and action of objects. In this next example, consider a *Navigator* with the simple behavior of being *At* a *Location* or *Going* from one *Location* to another. The NDDL model given in Figure 4 extends the path network model to include the class and predicates required for navigation.

```
// Include prior model file, re-using the definition of Location
#include "classes.0.nddl"

// Declare a class for describing basic Navigation state and action.
class Navigator extends Timeline {
   // A predicate for the state of being at a location for a period of time
   predicate At{
    Location location; // Paramater is Location
   }

   // A predicate describing the action of going from one place to another
   predicate Going{
      Location from, to;
      // Intrinsic to the definition of this predicate is that we cannot go to where we are!
      neq(from, to);
   }
}
```

Figure 4: Navigation support

Note that declaration of arguments to a predicate follow patterns of variable declaration and allocation already described, including implicit and explicit base domain specifications. However, the *new* operator is not permitted. Note also that there may be cases where relationships exist among predicate arguments. These are examples of internal token relationships discussed in 3.2. They are defined in the predicate declaration itself as constraints in the same way that constraints were introduced in prior sections. This example prohibits going to a location one is presently at by posting a constraint in the body of the predicate declaration which imposes the relation that its arguments are not equal (*neq*). Finally, notice that the *Navigator* class extends a *Timeline*. Thus it inherits semantics of mutual exclusion described in the EUROPA Overview.

The NDDL listing in Figure 5 uses NDDL to construct a partial plan with:

1. A set of Locations
2. A single instance of a Navigator
3. 2 tokens, i.e. instances of predicates, for being at a *Rock* and later being at a *Lander*.

```
// Include the model
#include "classes.1.nddl"
```

```
// Allocate instances for this plan database
Location Hill = new Location("Hill");
Location Rock = new Location("Rock");
Location Lander = new Location("Lander");

// Allocate a navigator using the default constructor.
Navigator nav1 = new Navigator();

// Indicate that the database is closed - no new objects can be created
close();

// Now allocate instances of predicates i.e. tokens

// First token
goal(nav1.At t0);

// Specify that it is located at the Rock
t0.location.specify(Rock);

// Specify that it is at the rock from timepoint 0. No end time is indicated.
t0.start.specify(0);

// Second token
goal(nav1.At t1);

// Specify that it is at the Lander
t1.location.specify(Lander);

// Specify that it is at the Lander from timepoint 1000.
t1.start.specify(1000);

// Impose a temporal distance of between 20 and 100 time units between the end of the first token
// and the start of the second.
temporalDistance(t0.end, [20 100], t1.start);
```

Figure 5: A partial plan for navigating from Rock to Lander

A number of items are worth noting:

1. Commands are executed in the order they are defined.
2. A *goal* keyword is used to introduce a token. Such a token is immediately *active.* Specifically, the base domain of its built-in *state variable* will be restricted to ACTIVE, thus preventing *merge* and *reject* operations.
3. An instance of *Navigator* is allocated prior to creating any token. If a token of the *Navigator* class were created prior to allocation of an object, then the base domain of the built-in object variable would be empty. This would yield an inconsistent state and is not permitted for tokens.
4. Unlike object member variables, token parameter variables are not initialized specially during allocation. Therefore, the base domain is derived from the defaults in the predicate declaration. Further restriction is possible using the assignment statement.
5. A scoping operator is used to access variables of tokens. This example accesses built-in variables (e.g. *start*, *end*) as well as user-defined parameter variables (e.g. *location*).
6. The constraint *temporalPrecedence*[3] is used to impose a temporal constraint between the time points of respective tokens.

Inheritance

Inheritance is a second method of re-use in NDDL[4] designed to make models more compact, and modeling less laborious and error-prone. The goal is that NDDL libraries may be developed that can be applied on many applications in a common domain (e.g. robotic control). The *Navigator* example incorporated class inheritance using the *extends* keyword (We saw an example already in Figure 4). There are no scope qualifiers at this time so all member variables and predicates of a class are publicly available outside the class and by subclasses. The general pattern is:

```
class DerivedClass extends SuperClass { ? }
```

An additional language construct, the keyword *super*, is provided to invoke base class constructors. It is only usable in the body of a constructor.

Figure 6 illustrates the application of inheritance. Note that in the derived class it is possible to add new member variables, new predicates, and additional parameter variables to inherited predicates. In addition, further restrictions can be placed on inherited predicates. In this example a more restricted base domain is provided for parameter *Foo.arg1* and a constraint is imposed between *Foo.arg1* and *Foo.arg2*.

```
// Declare a simple class
class Foo {
   // Declare member variables of primitive types
   int arg1;
   float arg2;
   bool arg3;

   // Declare a constructor with no arguments to conduct default initialization
   Foo(){
      arg1 = [0 10];
      arg2 = 0.0;
      arg3 = false;
   }

   // Declare a constructor with 2 arguments, default the 3rd
   Foo(int _arg1, float _arg2){
      arg1 = _arg1;
      arg2 = _arg2;
      arg3 = false;
   }

   // Declare a constructor to explicitly initialize all arguments
   Foo(int _arg1, float _arg2, bool _arg3){
      arg1 = _arg1;
      arg2 = _arg2;
      arg3 = _arg3;
   }
}

// Declare a subclass
class Bar extends Foo {
   string arg4; // Add another argument

   Bar(){
      // Must explicitly invokes superclass default constructor!
      super();
      arg1 = [5 10];
```

```
      eq(arg1, arg2); // XXX is this right?
      arg4 = "empty string";
   }

   Bar(string _arg4){
      super(); // Explicitly invoke superclass default constructor  super();
      arg4 = _arg4;
   }

   Bar(string _arg4, int _arg1, float _arg2, bool _arg3){
      super(_arg1, _arg2, _arg3); // Invok specific superclass constructor with arguments
      arg4 = _arg4;
   }
}

// Allocate instances
Bar bar1 = new Bar();
Bar bar2 = new Bar("hello");
Bar bar3 = new Bar("goodbye", 10, 20.6, true);
```

Figure 6: An example of class inheritance which illustrates use of extends and super keywords.

Semantically, inheritance defines an *isA* relationship between classes such that all instances of a derived class can be treated as instances of a parent class. The converse is not true. This has implications for populating object variables, which in turn impacts merging. For example, consider the following:

```
Base b1 = new Base(1);
Base b2 = new Derived(2);
Base b3 = new Base(3);
Derived v0;
Base v1;
```

The above will result in 3 objects named *b1*, *b2*, and *b3*. It will also result in 5 variables. Variables *b1*, *b2* and *b3* will all have singleton domains. Variables *v0* and *v1* will have domains of *{b2}* and *{b1, b2, b3}* respectively.

Resources

EUROPA has some pre-defined multi-capacity numeric resources from which it is often useful to inherit:

- **Resource**: Represents any resource and has a single predicate, `change`, used to represent additions or subtractions from the resource.
- **Reusable**: Represents a renewable/reusable resource such as labor. Has a `uses` predicate whose `quantity` field indicates the amount of resource used; the resource is used for the duration of the `uses` token and then is available again.
- **Reservoir**: Represents a consumable resource such as battery power. Has `produce` and `consume` predicates to represent increases and decreases, respectively, of the resource.
- **UnaryResource**: A reusable, discrete, single-capacity resource and is efficiently supported using a timeline rather than independent consume and produce transactions.

There are usually constraints on how resources can be used and produced. The following member variables (all floats) represent those constraints and can be initialized in the class constructors or updated later:

- *initialCapacity*: The resource's starting level.
- *levelLimitMin*: The resource's minimum level.

- *levelLimitMax*: The resource's maximum level.
- *productionRateMax*: The resource's maximum production rate - the most that can be added per time unit.
- *productionMax*: The resource's maximum production - the most that can be added at one time.
- *consumptionRateMax*: The resource's maximum consumption rate - the most that can be removed per time unit.
- *consumptionMax*: The resource's maximum consumption - the most that can be removed at one time.

See PLASMA/trunk/src/PLASMA/Resource/component/NDDL/Resources.nddl for the complete NDDL specification of the resource classes.

Here is some code that shows a *Battery* class that inherits from *Resource* and sets the initial capacity and minimum and maximum levels in the constructor:

```
class Battery extends Resource {
  Battery(float ic, float ll_min, float ll_max){
    super(ic, ll_min, ll_max);
  }
}
```

Here is a predicate that consumes battery power:

```
Bulb::lightOn{
   starts(Battery.change tx);
   eq(tx.quantity, -600); // consume battery power
}
```

Internally, EUROPA uses maximum flow and incremental maximum flow (the default) algorithms to compute resource envelopes. These are documented in here (Muscettola, 2001) and here (Muscettola, 2004).

Rules

This section describes the facilities in NDDL for describing internal and external relationships between tokens and between token variables. These relationships were described at a high-level in Planning Approach.

Basic Rule Definition

The absence of rules governing interactions between *Navigator.At* and *Navigator.Going* in Figure 13 lead to an incomplete partial plan which had to be completed with explicit specification of an appropriate linkage. To enable a planner to bridge this gap automatically, model rules can be added to indicate the required relationships. Figure 7 presents a listing which provides the rules to accomplish this.

```
#include "classes.2.nddl" // Reuse the definitions

// Define compatibility rules for the At predicate
Navigator::At{
   // Allocate a token of the Going predicate as a slave.  Impose a meets temporal relation such that t
   meets(object.Going successor);
   // Constrain the slave so it is going from this location
   eq(successor.from, location);
   // Allocate a token for the going predicate as a slave.  Impose a met_by temporal relation such that
   met_by(object.Going predecessor);
   // Constrain the slave so that it is going to this location
```

```
   eq(predecessor.to, location);
}

Navigator::Going{
   meets(object.At successor);
   eq(successor.location, to);

   met_by(object.At predecessor);
   eq(predecessor.location, from);
}
```

Figure 7: Bridging the gap with domain rules.

Rules are written for a specific predicate, and are then applied to all tokens of that predicate that are ACTIVE[5]. There is no limit to the number of rules that can be defined on a predicate. Different rules might be written to reduce individual rule complexity and/or to make rules more modular. Rules can be all in a single file or distributed in many files. However, each rule must be defined in a single file. Rule declaration can be interleaved with class declaration as long as classes and predicates referenced in a rule are already declared. The syntax for introducing a rule follows:

```
ClassName::PredicateName {<rule-body>}
```

Slave Allocation and Temporal Relations

A slave token is most commonly introduced using a temporal relation. The set of all temporal relations is based on the Allen Relations of (Allen, 1983) and were presented in Figure 2 on the Planning Approach page. For example, the statement:

```
meets(Going successor);
```

allocates a new slave token which will be referenced in the scope of the rule with the label *successor*. It further imposes a constraint that the *end* time of the master token (i.e. the token on which the rule is being applied) is concurrent with the *start* of the successor. This can also be stated more verbosely using the any keyword and an explicit constraint:

```
any(Going successor);
concurrent(this.end, successor.start);
```

In the above, *this* is used to refer to the master token context and can be used to disambiguate variable names. All Allen relations are turned into primitives in this way using general slave allocation and explicit constraints during model compilation. Other valid constructs for token allocation or specification of temporal relations between tokens are:

```
any(Foo.p); // Allocates a slave of a predicate p with no label on a class Foo.
any(var.p); // Allocates a slave of a predicate p with no label on a class given by the type of var and
  // object in the domain of var, where var is an object variable.
Any(Foo.p t1); // Unconstrained slave t1
Any(Foo.p t2); // Unconstrained slave t2
t2 meets t1; // An Allen relation between 2 slaves
this meets t2; // An Allen relation between the master and the slave
```

The supported binary temporal relations[6] are :

Relationship	Implied Constraints
before	precedes(origin.End, target.Start);
after	precedes(target.End, origin.Start);
meets	concurrent(origin.End, target.Start);
met_by	concurrent(origin.Start, target.End);
equal or equals	concurrent(origin.Start, target.Start); concurrent(origin.End, target.End);
contains	precedes(origin.Start, target.Start); precedes(target.End, origin.End);
contained_by	precedes(target.Start, origin.Start); precedes(origin.End, target.End);
paralleled_by	precedes(target.Start, origin.Start); precedes(target.End, origin.End);
parallels	precedes(origin.Start, target.Start); precedes(origin.End, target.End);
starts	concurrent(origin.Start, target.Start);
ends	concurrent(origin.End, target.End);
ends_after	precedes(target.Start, origin.End);
ends_before	precedes(origin.End, target.Start);
ends_after_start	precedes(target.Start, origin.End);
starts_before_end	precedes(origin.Start, target.End);
starts_during	precedes(target.Start, origin.Start); precedes(origin.Start, target.End);
contains_start	precedes(origin.Start, target.Start); precedes(target.Start, origin.End);
ends_during	precedes(target.Start, origin.End); precedes(origin.End, target.End);
contains_end	precedes(origin.Start, target.End); precedes(target.End, origin.End);
starts_after	precedes(target.Start, origin.Start);
starts_before	precedes(origin.Start, target.Start);
any	None

Note that the same keywords can be used to express Allen constraints using different syntax. Such constraints do not involve slave tokens and are documented in the Constraint Library Reference.

Variables and Constraints

The material presented above for handling variables and constraints at the global level applies equally in the context of rules. All variables accessible in a rule context can be incorporated into constraints in precisely the same manner as seen earlier. This was been illustrated in Figure 16 where the equality constraint was used within rules for *At* and *Going* predicates. The scope for accessing variables in a rule includes:

1. All token variables. All built-in and user-defined predicate parameters for the master token are accessible without qualification, or with explicit qualification using the keyword *this*.
2. All global variables. All variables declared outside the scope of a class or predicate are considered global.
3. All slave variables. All built-in and user-defined parameters on any slave token introduced in the rule and labeled. The label is required to reference a slave?s variables.
4. Local variables. It is possible to declare local rule variables in the same way that one declares global variables. However, it is not permitted to use the *new* operator within a rule. Local variable names must not conflict with global variable names or variable names already defined on the token.

It is worth mentioning that the composition relationships permitted in class declarations can be traversed using the ?.? operator.[7] For example, suppose the following class structure:

```
class A{int i = [1 10];}
class B{A a; B(){a = new A();}}
class C{B b; C(){b = new AB();}}
Given the above structure, any variable declared of type ''C'' can be navigated to obtain a structurall
C c; // A local variable of type C. Could instead be a variable on a token, a global, etc.
int j;
eq(c.b.a.i, j); // Traverse the object structure to access a specific variable.
```

Guards

It is possible to include extra qualifications to guard introduction of slaves or constraints in a rule. This applies wherever rules or rule elements are conditional. A guard is placed around a block of statements in a rule using the if keyword. For example:

```
Bool b;
if(b == true){
        <rule statements>
}
if(b == false){
        <rule statements>
}
```

The above NDDL rule fragment includes a declaration of a local variable and then provides 2 guarded branches based on values of that variable. Guards are only evaluated when the variable in question is specified to a singleton. Only the test for equality is currently supported. Other common forms for expressing conditional statements such as *else* or *switch/case* combinations are not currently supported. One can also have:

```
if(v){
        <rule statements>
}
```

This has the semantics of firing if and only if the variable v is specified to a singleton value. It is not possible to specify multiple conditions in a single guard statement at this time. Instead, guards can be nested. For example:

```
if(a==1){
    if(b==2) {
        if(c==3){..}
    }
}
```

Filtering

The navigation model presented in Figure 16 had a rather glaring deficiency: it was permitted to go from any location to any other location whether a path existed or not. Recall that the model included a *Path* class which is a link between two locations and a cost associated with that link. It is desirable to restrict navigation to stay on defined paths in a problem. To accomplish this, a rule can be added on the *Going* token which states the equivalent of:

There exists some path such that its goes from my source location to my destination location.

In NDDL this could be another rule, or an addendum to the prior rule. As a new rule:

```
Navigator::Going{
```

```
Path p;
eq(p.from, from);
eq(p.to, to);
}
```

Figure 8 illustrates the semantics in terms of variables and constraints for this seemingly simple construct. It shows a simple path network with 5 locations and 5 path links.

Figure 8: An illustration of the underlying semantics for existential quantification.

A proxy variable is created for each referenced member of the path variable p (i.e. *p.from* and *p.to*). Each proxy variable is populated with the union of respective member values over all paths. A special constraint is created to maintain synchronization as domains are restricted in both directions. It is the proxy variables which are actually used in the constraints explicitly defined in the model. In the example only an equality constraint was used. However, the proxy variable formulation allows any constraint to be used in filtering providing a powerful technique for expressing existential quantification constraints. Only members that are singletons or enumerations can be used for filtering.

Iteration

Consider a domain rule where all hatches must be sealed before a submarine can submerge. This could be modeled with a *submerge* action and a rule requiring all hatches to be in a *closed* state prior to and during submerging. It might be cumbersome and/or impractical to impose this relation explicitly on specific hatch instances. To address this, NDDL includes a *foreach* structure permitting universally quantified relationships over abstract sets, as shown here:

```
Submarine::submerge {
    Hatches hatches; // Local variable populated with teh set of all hatches
    foreach(hatch in hatches) {
        contained_by(hatch.closed);
    }
}
```

The general form is:

```
foreach(label in set){
        <rule statements>
}
```

The set must be closed and its contents cannot be further restricted through plan refinement. This is a subtle but important prohibition. Since a rule statement within a single iteration might include allocation of a constraint or token, a restriction on the set over which the iteration applied would then imply that a constraint or token be removed. This is non-monotonic. Instead, a lock constraint is imposed when imposes a lock on the set over which the iteration occurs once it is fired.

A further complication arises since the semantics of universal quantification do not require existence of an element over which the quantification holds i.e. if there were no hatch on the submarine the rule should still be satisfied. This is problematic since an empty variable typically indicates an inconsistency. To address this NDDL allows a variable declaration to be marked as a filter instead of a true variable of an underlying CSP. The semantics of this are that it cannot be decided by a solver and it can be emptied without implying an inconsistency. The marker keyword *filterOnly* is added at the end of a variable declaration. For example:

```
Hatch hatches filterOnly;
```

The NDDL Transaction Language[8,9]

NDDL includes procedural extensions, referred to as the *NDDL Transaction Language*, to operate on the plan database and thus initialize and/or modify a partial plan. A design goal of the NDDL transaction language is to provide syntax and semantics closely related to the use of NDDL elsewhere for class, predicate and rule declaration. However, the NDDL transaction language pertains exclusively to run-time data. It is referred to as a transaction language since a set of statements in this language form a procedurally executed sequence of atomic operations on the plan database, which stores an instance of a partial plan. Each statement of the language is thus directly translated into one or more operations available through the *DbClient?* interface. The NDDL transaction language has many applications. The most common one is the construction of an initial partial plan as an input to a solver. A second important application is to log transactions on the database for later replay. This is useful for copying a database, and for reproducing a state found through planning in a direct manner without having to search. It is also a potentially very useful integration mechanism for pushing updates to the database from external systems. Many of the available operations have been touched on in prior examples. This section presents the full language. The operations covered are:

- **Global Variable Creation**: Declare and define a global variable which can be referenced by subsequent statements in a transaction sequence or from within the model.
- **Constraint Creation**: Create an instance of a given registered constraint between any variables in the transaction sequence context.
- **Object Creation**: Allocate instances of any class declared in the model, using any of the declared constructors.
- **Type Closure**: Indicate that no new instances of a class shall be created. Unless explicitly closed, all types remain open.
- **Token Creation**: Allocate instances of any predicate declared in the model.
- **Variable Assignment and Un-assignment**: Commit to specific values or domains for any variable in the transaction sequence context and to withdraw that commitment.
- **Token Activation and Deactivation**: Force a token to be active and the means to retract from this state.
- **Token Merging and Splitting**: Force a token to be merged, and the means to retract from this state.
- **Token Rejection and Reinstating**: Force a token to be rejected, and the means to retract from this state.

We have already covered variable and constraint creation using the NDDL transaction language. We showed:

- Declaration of a variable with a default base domain - e.g. `int i; Colors colors;`
- Declaration of a variable with an explicit and restricted base domain - e.g. `int i = 6; int j = [10 40]; Colors colors = Blue;`
- Declaration of an object variable and allocation of an object - e.g. `Foo f = new Foo();`
- Declaration of one variable by assignment of the contents of another - e.g. `Foo b = f;`
- Allocation of a constraint - e.g. `eq(i, j); neq(j, k);`

We also introduced predicates, composition and inheritance and showed that constructors with arguments can be invoked explicitly, passing in values, domains or variable references - e.g. `Bar b = new Bar(f);` We also showed that tokens can be created using the goal and mandatory keywords. This section revisits some of these operations and discusses some new material.

Type Closure

Most commonly, all objects are instantiated in the Plan Database before any tokens or constraints are added. Once all the objects are created, it is typical to indicate to the database that no more objects can be added. The most straightforward way to accomplish this is with the statement: `close(); // Close the plan database. No new objects can be added` Under certain circumstances, it may be preferable to allow objects of a particular type to be created throughout the lifetime of the plan database. The general form of such a statement is `class.close();` to prohibit further instantiation of objects of type class. Here are some examples:

```
Rover.close(); // Close the database to new Rover instances (and any subclasses).
Timeline.close(); // Close the database to new Timeline instances (and any subclasses).
```

Any attempt to allocate an instance of a class that has been closed will result in an error. By default, all types remain *open* until they are explicitly closed. Unless you have specialized requirements for *Dynamic Objects*, you should always insert a `close();` statement before creating any tokens. There is no operation to open a class once it has been closed.

Creating Tokens

There are 2 ways to introduce a token into the plan database using NDDL transactions. The first employs the *goal* keyword. It has the form *goal(objectScope.predicate [label]);* where *objectScope* designates the set of objects to which this token can be assigned and predicate designates the particular predicate to be created. A label is optionally used if later NDDL statements wish to refer to the instance to be allocated. For example, consider the statement: `goal(Navigator.At); // Allocates an anonymous active token which can be assigned to any instance of Navigator` This statement results in a new token in the plan database which will be in the *active* state. The object variable of the new token will be populated with the set of all instances in the *Navigator* class present in the database at the time of token creation. Other examples include:

```
goal(nav1.At); // Allocates an anonymous active token with a singleton object variable == nav1
goal(nav1.At t0); // Allocates a labeled active token t1 with a singleton object variable == nav1
```

Upon execution of this command, the state variable of the new token={ACTIVE}. This means that the token cannot be MERGED or REJECTED. The second method uses the *rejectable* keyword. The keyword *rejectable* is identical in form to the goal keyword. The only difference is that the resulting token is in an *inactive* state. Upon execution of this command, the state variable of the new token is *{ACTIVE, REJECTED}*. This means that we can *activate*, or *reject* the token.

Specifying and Resetting Variables

We have seen many example statements where the domain of values of a variable was restricted:

- On construction it is restricted to the set of all values in the default base domain for the type e.g. int i;
- On construction it is restricted to an explicit domain e.g. int i = [0 100];
- On construction it is restricted to a singleton value e.g. int i = 10;
- A constraint is posted on the variable relating it to another variable, domain or value.

The assignment operator ?=? restricts the base domain of a variable and is the preferred method of indicating domain restrictions in the initial state. It is more efficient than posting a constraint and makes a stronger commitment since base domain restrictions are not retractable. However, the NDDL Transaction Language also

supports *retractable* commitments on a variable. This is useful to record and replay operations on the database made by a solver, or some other client. There is an operational difference between setting a variable to a value, and constraining it to the same value since rules of inference which may be conditional on this value will not be evaluated until the value is specified. Therefore, a method is provided to *directly specify* the value of a variable as a *retractable* commitment and a companion operation is provided to withdraw the commitment. Here is an example transaction sequence (and tiny model) using *specify* and *reset*:

```
class Foo { // Declare a basic class with a simple predicate
    predicate pred{int arg;}
}

enum Color {Red, Yellow, Blue}; // Declare an enumeration

// Allocate objects and close the database
Foo f1 = new Foo();
Foo f2 = new Foo();
Foo f3 = new Foo();
close();

Foo allFoo; // Declare a variable. Contains all Foo instances initially
allFoo.specify(f2); // Specify it to a singleton
allFoo.reset();

goal(Foo.pred t0); // Allocate a goal token - initially active. Initially the object variable contains
t0.object.specify(f1); // Specify to a singleton. Demonstrates access to built-in token variable
t0.arg.specify(4); // Specify to a singleton. Demonstrates access to an explicit token parameter variab
Color color; // Allocate a variable which initially contains all colors.
color.specify(Red); // Specify to a singleton
```

Operations on Tokens

We have described the set of internal token states indicating their role and relevance in the plan database and the partial plan. It has been shown how tokens are introduced into the plan database using either *goal* or *rejectable* statements. The example below demonstrates the use of NDDL transactions to effect state changes in the token.

```
// Declare a basic class with a simple predicate
class Foo {
    predicate pred{}
}

Foo foo = new Foo();
close();
goal(foo.pred t0); // Initially active
rejectable(foo.pred t2);  // Initially inactive
t2.reject(); // Now rejected
rejectable(foo.pred t3);  // Initially inactive
t3.activate(); // Now Active
t3.cancel(); // Now inactive again
```

Token Relationships

In addition to the general capability of creating constraints among variables, NDDL supports special kinds of relationships to be stated among tokens. These relationships are based on the Allen Relations introduced described earlier in the context of model rules. The general form for specification of these relations is a binary relation:

```
tokenA relationName tokenB;
```

This imposes the specified relation *relationName* between *tokenA* and *tokenB* and is identical to the use of qualitative temporal relations presented in section 5.3.2. In addition, a very specific type of ordering relation, *constrain*, is supported which is equivalent to the before temporal relation in terms of its effect on the time-points of the variable. However, it is *object specific* and is used to also indicate an assignment of the tokens involved on the *Timeline*. Unlike the situation with temporal relations, this operation can be reversed using the *free* operation. These operations are the basis for resolving *threats*. The example below illustrates usage of these 2 operators.

```
// Declare a basic class with a set of simple predicates
class Foo extends Timeline {
predicate pred0{}
predicate pred1{}
predicate pred2{}
predicate pred3{}
predicate pred4{}
}

// Allocate 2 objects
Foo foo1 = new Foo();
Foo foo2 = new Foo();
close();

// Allocate active tokens. Initially the object variable of each will contain
// 2 values - foo1 and foo2
goal(Foo.pred0 t0); // Initially active
goal(Foo.pred1 t1); // Initially active
goal(Foo.pred2 t2); // Initially active
goal(Foo.pred3 t3); // Initially active
goal(Foo.pred4 t4); // Initially active

// Pass the same token for both arguments as a degenerate case to assign the token to the object.
// This will probably be deprecated.
foo1.constrain(t1,t1);

// Constrain t2 and t3 such that t2 before t3 and both assigned to foo2,
foo2.constrain(t2, t3);

// Constrain t1 and t4 to foo1.
foo1.constrain(t1, t4);

// Constraint t0 before t1 on foo1
foo1.constrain(t0, t1);

// Now free it to illustrate the reversal.
foo1.free(t0, t1);
```

Summary

This chapter has presented the means of describing problem domains and partial plans in E2 using the NDDL modeling language. NDDL provides a very high-level, declarative, object-oriented approach for specifying the types to be considered in a domain and the nature of the interactions among domain elements. It is an open language in the sense that it promotes extension by integration of plug-in constraints. Chapter 8 will further show how the expressive power of the language can be extended with customized base classes. Being able to describe the salient features of a domain of interest is a necessary condition for problem solving in that domain. However, it is not sufficient. Powerful problem representation must be complimented with effective algorithms to actually solve

problems. That is the topic of the next chapter.

---

1. DDL was the domain description language introduced in (??) and applied on the remote agent experiment (??).
2. Variables of primitive types, typedefs and enumerations are closed on construction. Classes are the only types that may remain open - meaning that the set of instances of that type (i.e. that class) may not yet be fully defined.
3. This is equivalent to *addEq* but is specifically used in cases where a temporal relationship is intended. Similarly, the constraint concurrent(x, y) is a temporal constraint that is semantically equivalent to *eq*(x, y) but allows specialized global propagation.
4. The first being composition.
5. Rules are not expanded until a token has been committed to the plan. The reason for this is mainly practical since the process of merging tokens would be more convoluted and expensive in time and pace and the benefits of doing otherwise are not clear. It is possible that rule expansion could be applied on inactive tokens in the future.
6. If the master were explicit they are all binary relationsbetween tokens.
7. '.' is a *member accessor* operator and can be applied to object variables and tokens.
8. In the current implementation, inconsistency is not an option whn processing transactions. It is a limitation imposed by the procedure which processes NDDL transaction sequences that teplan database must be consistent after each step. If an inconsistency is found, it will abort the program.
9. The NDDL language described in this section is converted to an xml-based language. It is not strictly necessary to use the NDDL syntax and one can work directly in XML if appropriate.